Measuring the success of a test automation project

By Subject7 and Matt Heusser

Beyond Faith

Return on Investment

A simple example: Automated deployments
A more difficult example: Automated regression testing

Measuring Intangibles

Measurement and Dysfunction

Putting It All Together

The majority of test automation projects we see have ill-defined or completely unstated goals. Often, they are less like projects and more like approved budget requests: A company hires a test automation engineer and presumes this will solve their problems, or pays for a tool in hopes of "saving money." Six months later, the savior employee is no longer with the organization and the magic tool is sitting on a shelf.

When goals are unclear, there is no method to measure success or failure. The inevitable results are wasted time, misspent money, and organizational reluctance to try again soon.

Most successful software projects begin with the end in mind. They look at the goal, find a way to value that goal, create a realistic plan, and, if the project is profitable enough, fund it. After all, executives are still looking for a return on investment, payback, alternatives, and the balanced scorecard to make that decision. Funding often includes a series of stage-gates with criteria that must pass in order to get to the next injection of cash.

A test automation project should be handled more like other successful projects.

Let's examine the challenges that come with measuring the value of a test automation project. We will explain how return on investment applies, give examples of how to gauge the effectiveness of changes, and show how you can define success for your test automation effort before you start. With those metrics in mind, you can select the right approach to your test automation project.
Beyond Faith

The simplest motivation for most test automation projects could be called "faith-based test automation," or just wanting an automation initiative because that’s what you’re supposed to have. For example, the core of what programmers do is write code, which can be to solve problems, or "automate things.” Ergo, automation is simply what programmers do.

Another common reason organizations start pursuing a test automation project seems to be that everyone else is doing it. Many pundits and authors work from a presumption that test automation is inherently good and teams must do it to be successful.

The third argument—perhaps the most compelling—is that automation is eating the world. Jobs such as telephone switchboard operator, elevator attendant, and bowling pin setter have disappeared in recent human memory and been replaced by technology. From the outside, manual human testing appears to be a simple job, so it seems as though it could also make a good candidate to be replaced by technology.

However, none of those arguments are evidence-based, or even reasoned. They are beliefs held with no proof, which means they won’t work well to convince others that your test automation project is a success—or even necessary.

You need tools to support measuring and modeling so that you can objectively prove what your test automation is doing. This way, you can decide for yourself if more automation is the right investment for your organization. Like most predictive tools, they are not perfect. They involve a fair amount of ambiguity and uncertainty. Still, they can help in the decision-making process.

The major measure we want to introduce you to is return on investment, or ROI.

Return on Investment

There is a perception that technical people are interested in tools, middle management is interested in process, and senior management is interested in financial results. There might be something to this. Technical staff who cannot understand or explain the impact of their ideas will likely fail to realize change at higher levels. Middle managers who need to predict outcomes may select measures with faulty presumptions that do not impact the bottom line.

Calculating return on investment gives senior managers an understandable foundation. Middle managers can explain it, and technical staff can appreciate and support it.

The formula for ROI is deceptively simple:

\[
\text{Return on investment as a percentage} = \frac{\text{Improved value} - \text{Cost of investment}}{\text{Cost of investment}} \times 100
\]
Consider spending $7,000 on an investment that added $10,000 worth of value.

\[
\text{ROI} = \frac{($10,000 - $7,000)}{7,000} = \frac{\text{($10,000 - $7,000)}}{7,000}
\]

That is a 42% return on investment. We find that businesses like to keep that return to a one-year period, or two years at most. Beyond that, the models get more complex and include things like the time value of money and return on an annualized basis.

Doing the math, for software, the "improved value" variable can sometimes be difficult to quantify. When the idea of value is unclear, the easy thing to do is calculate cost savings alone.

Cutting costs is a strong temptation, but this can lead to greater challenges. Most cost reductions in software involve downsizing, outsourcing, and offshoring, usually resulting in lower productivity and often slower delivery.

In order to justify cost, we need a way to calculate value. Here are a pair of examples of how this might play out.

Example 1: Automated deployments

Team A moves code to production once every month, after a one-week regression process. This process includes three or four cycles of testing, documenting bugs, and then retesting.

The reason testing happens so late is that the deployment process is difficult. Tickets often get passed back and forth and involve a fair number of manual steps. The impact of the manual effort can be measured in days of productivity.

Now consider that the team asks a developer to create a script that automates deployment for any given branch in version control. This process takes a few hours, and the programmer suggests deployment time to be every other week, thereby reducing the previous manual effort expended.

We can measure the impact of this change using our ROI model. Previously regression took 5 days at the end of each calendar month. Now regression testing takes only a day or two each month. That's team time recovered to work on new development. Assuming there are still some delays and surprises, let's say that after a year, 9% of the team’s time is recovered and it takes 5% of the team’s time to implement the automation solution.
That is an 80% return on investment in just the first year.

In our experience, cutting the regression process by more than half in exchange for 5% of the team’s overall effort is a reasonable cost. For a 10-person team, that is one person’s effort for six months. The increased efficiency is possible through automation.

Example 2: Automated regression testing

Before moving code to production, Team B spends the last day of a two-week sprint doing regression testing. In theory, if the team eliminates regression testing, they could spend that saved time developing software.

The team has 10 people and hires a full-time test automator. This allows the original team of 10 to go 10% faster because they no longer need to do regression testing themselves. This comes with a cost of 10% added to the development team.

Calculating the value here forces the organization to look at value the software groups adds. If we say the group merely pays for itself or is required to “keep the lights on,” then the return on investment is zero, as the improved value matches the cost.

However, this assumes the test automation software is free and regression testing ends automatically at the end of sprint one. This is unlikely. Instead, the new automator may spend six months or more before the toolset is ready. This means at least six months before there is a significant reduction in the work the team faces. This gives us a -10% ROI in year one, and the potential to break even in year two.
If we model the software itself as a modern business investment returning 10%, then the value is our 10% more velocity. With the associated level of cost, it makes more sense to simply hire one more developer, skipping the six months of negative ROI. If regression testing were more than 10% of the effort, the numbers would look more compelling.

These examples start to illustrate how measurements and metrics can help determine decisions about a test automation project, but these models are still too simple. They fail to account for failures or maintenance effort. Let’s examine some more realities of software projects.

**Measuring Intangibles**

There are plenty of benefits not included in the previous examples.

Assuming the automation runs overnight, it may be able to find a defect within a day of the defect being introduced. If the automation runs continuously, a defect may be found shortly after it is introduced in the build. In both cases, the programmer who introduced the problem may jump in and fix it.

However, when testing happens as a phase independent of development, days or even weeks may pass before a defect is found. Then the defect will need to be documented and triaged. It will be assigned to a programmer who may not be familiar with the change that introduced the problem, and even if they are familiar, it is likely going to take time to refamiliarize themselves with the code.
If the bug is found in production, it will be even more expensive to fix. A conservative estimate of the amount of thrash from "bug work" might be 5%. That thrash can push out new development. One of the easiest improvements to make is to fix bugs faster (or have less of them) and spend that reclaimed time on new development. The problem is that most organizations do not have the measurement systems in place to track such work.

Other intangibles include the ability to attract and hire more quality talent, less employee turnover, higher morale, less mandated overtime, and better collaboration between development, testing, and management.

At the same time, the strategies above assume the job the humans will do and the job the computer will do will be the same. It is possible the tool finds different kinds of defects. Done well, this can optimize performance between the two. Done poorly, automation tools can ignore entire categories of risk.

Here are a few elements of software value that can be difficult to turn into a dollar value but may be worth considering.

*Quality of bugs found.* If the test automation tool doesn't find bugs that matter, it will provide less value. If it can find bugs humans miss, that will provide more value, but only if those bugs are fixed and the customer experience improves.

*Frequency of releases.* Quarterly releases provide no value until the beginning of month four. Increasing the release frequency delivers value at a faster rate. Most of the organizations we work with struggle to quantify the value of the new software delivered. They may be able to explain the increase in terms of velocity, but that is not the same thing as dollars.

*Time to recovery.* Test automation can lead to fixes that are rolled out faster, which can improve the customer experience. Time-for-a-bug-to-live may be easy to pull out of a system like Jira, but many bugs will not be fixed, and it’s difficult to assign a monetary value.
Customer experience. In a software as a service (SaaS) group, the measurements are customer retention, growth, and monthly cost. Separating how the automation adds to that can be difficult. Customer experience surveys can help bridge that gap.

Time horizon. We suggest a one-year time horizon for most of these efforts. In some cases, a longer or shorter horizon can be helpful. If a test automation effort can't pay for itself within a year, if payback isn't done within two years, and if profit isn't clear within three, it may be time to pause and reflect.

Trade-offs. Most of us are familiar with the idea of reducing quality to deliver the software a little earlier. A balanced scorecard looks more like a dashboard, including things commonly traded, such as code complexity, velocity, and test coverage. Having a variety of measures makes it harder to "game" the results.

Measurement and Dysfunction

The pure math of ROI seems so easy: Just measure the increase in productivity, subtract the cost (time and cash), and calculate the value. But before we endorse this idea, we need to address the issue of dysfunction.

Measuring productivity is hard, so many organizations measure velocity. Likewise, getting a firm measure of quality is hard, so many organizations measure bug counts. The problem is what we are measuring is not what we are actually interested in. James Bach, a well-known software tester, pointed out that the difference between what we want to measure and what we actually measure is something that can be exploited.

Teams that measure velocity in points will have an incentive to overestimate stories. The team can break those stories into tiny, bite-sized pieces so that the number of stories delivered, or the velocity, goes up—even though productivity might not.

The same thing can happen with counting bugs. Staff can argue something is "not a bug." They might connect bugs that appear similar as related bugs in a single bug report. They might argue defects are actually features or change requests. Sometimes defects are not entered into the tracking system and instead get passed by notes, IM, or email.

Some organizations intentionally create conflict between programmers and testers by rewarding testers for finding bugs and punishing developers for those same bugs. Some have even sent testers to the movies so no bugs will be reported.

There are many bad measurement systems for value, quality, and productivity in software. Measurement done badly often causes more harm than good.
Putting It All Together

The pursuit of test automation without clear goals or definition of success is risky. With no meaningful measurements, software teams are likely to become frustrated by attempting to meet targets that are unreasonable, if not impossible.

The easy measurements, such as bug counts, are likely to be the easiest to game and the least meaningful. Complex measures, like customer satisfaction, can be expensive and hard to track. And even a well-balanced scorecard will have gaps.

To gain more useful information around your test automation project, begin with the end in mind. Have a plan for how the software test process should be changed.

That could include the elements of value described above, such as tightening or eliminating the regression-test window and deploying software more frequently. Look for fewer defects following each release. Aim for defects to be closed sooner after each release than the one before. Target customer satisfaction in particular.

At least a few of these objectives could be defined as SMART goals: Specific, Measurable, Achievable, Relevant, and Time-bound. Those are your test cases for your automation project.

Also be sure to be realistic when looking at the investment involved in automation. Look beyond the dollar outlay for the tools and keep in mind the time, effort, and lost opportunity cost around training, building, and ramping up the effort. Can the organization achieve the goals with the amount they are willing to invest? Is that investment the right one to make right now?

If the answer is yes, use the plan to manage the effort like a project. Look at your objectives and key results.

These tools can help you set meaningful measures for the success of your test automation project.